ISSN 1561-5359. Штучний інтелект, 2017, № 2

УДК 004.415.53+004.416.2

В.А. Волков, А.В. Колчин, А.А. Летичевский, С.В. Потиенко
Институт кибернетики имени В.М. Глушкова НАН Украины, Украина
пр. Академика Глушкова, 40, г. Киев, 03680, МСП

ОБЗОР СИСТЕМАТИЧЕСКИХ МЕТОДОВ АВТОМАТИЧЕСКОЙ ГЕНЕРАЦИИ ТЕСТОВЫХ ДАННЫХ ПО ИСХОДНОМУ КОДУ ПРОГРАММ

V. Volkov, A. Kolchin, A. Letychevskyy, S. Potiyenko
V.M. Hlushkov Institute of cybernetics NAS of Ukraine, Ukraine
40, Academician Hlushkov av., Kyiv, 03680, MSP

A SURVEY OF SYSTEMATIC METHODS FOR CODE-BASED TEST DATA GENERATION

Обзор включает описание распространенных типов покрытия и современных методов генерации тестов из кода императивных программ. Особое внимание уделено методам, в основу которых положена техника символьного выполнения. Обсуждаются преимущества различных подходов и недостатки автоматического порождения тестов в целом.

Ключевые слова: тестирование, покрытие, редукция пространства поиска.

The survey includes description of prevailing coverage types and modern methods of code-based tests generation for imperative programs. Special emphasis is made for methods, which are based on symbolic execution technique. Advantages of different approaches and disadvantages of the automatic tests generation as a whole are discussed.

Key words: testing, coverage, state-space search reduction.

Введение

Тестирование – самый используемый способ оценки качества в промышленной программной индустрии [1]. Методы автоматизации построения тестов на основе кода стали активно исследоваться в 1970-х годах [2], с тех пор создано много различных подходов к данной проблеме. Их можно классифицировать на основании принципов, положенных в основу алгоритмов:

- Стохастические (random) методы порождают тестовые данные случайным образом. Преимущества в производительности, простоте реализации и внедрения. К недостаткам можно отнести плохое покрытие, большую степень избыточности.
- Комбинаторные методы основаны на переборе значений входных параметров тестируемой программы. Обычно перебор всех возможных комбинаций невозможен, (например, сложение двух 4-хбайтовых целых допускает 2^{64} вариантов), поэтому прибегают к ограниченным переборам некоторых подмножеств.
- Методы поиска оптимального приближения (search-based) используют функцию приближения для измерения «дистанции» порожденной популяции тестов от искомых тестовых целей покрытия, и стремятся ее минимизировать.
- Систематические методы извлекают условия выполнения путей программы на основании некоторого анализа кода, решают эти условия и получают тестовые данные, которые направляют выполнение программы по этим путям.

Такая классификация довольно грубая и не исчерпывающая, часто системы реализуют гибридные и эвристические подходы. В данной работе приводится обзор современных систематических методов генерации тестов по принципу «белого ящика» на основании кода тестируемой системы.

Обзор типов покрытия кода

К тестовому набору выдвигаются требования обеспечения некоторого покрытия, при котором оно часто оценивается как отношение количества покрытых элементов к общему числу искомых элементов тестируемой программы. Далее приведены самые распространенные типы покрытия [1, 3]:

- покрытие функций (function coverage) требует, чтобы были вызваны все функции (подпрограммы);
- покрытие операторов (statement coverage) требует выполнения всех операторов;
- покрытие ветвей (branch coverage) требует выполнения всех управляющих структур, таких как операторы условия if (как выполнимого, так и невыполнимого случая) и выбора case (всех возможных вариантов);
- покрытие решений (decision coverage) требует покрытия условных операторов, как и в случае покрытия ветвей, а также рассматривает присваивание булевым переменным, например, для «`a = (b ∨ c)`»: покрытие будет достигнуто тестами, в которых переменная `a` будет принимать как истинное, так и ложное значение;
- покрытие предикатов в условиях (condition coverage) требует, чтобы все булевые подвыражения условий принимали как истинные, так и ложные значения;
- модифицированное покрытие условий/решений (modified condition/decision coverage) требует, чтобы каждый предикат в условии принимал как истинное, так и ложное значение, при этом влиял на выполнение всего условия;
- множественное покрытие условий (multiple condition coverage) требует покрытия всех возможных комбинаций (всей таблицы истинности условия);
- покрытие значений параметров (parameter value coverage, PVC) требует, чтобы методы, принимающие на вход параметры, вызывались со всеми распространенными типичными значениями (например, для строковых типов это null, пустая строка, пробел/табуляция/перевод строки, правильная строка, неправильная строка, строка однобайтных символов, строка двухбайтных символов. Тестирование только одного варианта может привести к 100% покрытию строк кода, однако только 14.3% (1/7) покрытию PVC);
- покрытие входов-выходов (entry-exit coverage) требует выполнения всех возможных вызовов и возвратов функций;
- покрытие линейных участков до перехода по потоку управления (JJ-Path coverage);
- покрытие циклов (loop coverage) требует, чтобы все циклы исполнялись 0,1,..N раз;
- покрытие потока данных (data-flow, или def-use coverage) требует, чтобы каждое присваивание переменной и использование этого значения было достигнуто и исследовано.

Метод «белого ящика» включает оценку полноты покрытия с помощью мутационного подхода. «Мутантом» называется незначительно модифицированный (в местах наиболее вероятных ошибок) исходный код. Качество покрытия тестами оценивается как способность выявления мутантов [4].

Рассмотрим детальнее критерии покрытия условий и потока данных.

Покрытие условий. Пример 1. Предположим, что необходимо протестировать следующий участок кода, представленный условным оператором «`if((A∨B)∧C)`», где `A`, `B` и `C` представляют атомарные логические выражения (т.е. не раскладываются на другие подвыражения). Для обеспечения покрытия условий, `A`, `B` и `C` должны принять значения «истина» и «ложь» хотя бы однажды при выполнении тестов, например, таких: (1) `A`=истина, `B`=истина, `C`=истина; (2) `A`=ложь, `B`=ложь, `C`=ложь.

Для обеспечения покрытия решений, условие ((A ∨ B) ∧ C) должно так же принимать значения «истина» и «ложь» хотя бы однажды. Покрытие достигается теми же двумя тестами. Критерии условий и решений не гарантируют покрытие всех атомарных условий, потому что во многих случаях некоторые предикаты «закрыты» другими: например, с помощью только двух предложенных тестов невозможно знать, какой именно предикат повлиял на результат вычисления полного условия.

В свою очередь модифицированное покрытие условий/решений (MC/DC) требует, чтобы каждый предикат условия принимал один раз значение «истина» и один раз «ложь», и при этом единолично влиял на результат полного условия. Это достигается парой тестов, в которых изменение значения только одного атомарного условия будет изменять также значение всего условия. Таким образом, критерий MC/DC значительно сильнее, чем критерии покрытия условий и решений. На практике, для условия, содержащего *n* атомарных логических условий, необходимо найти как минимум *n+1* тест для обеспечения покрытия MC/DC. В нашем примере предикатов всего три, мы можем, например, выбрать такой набор из четырех тестов:

1. A = истина, B = ложь, C = истина; условие «(A ∨ B) ∧ C» выполнимо
2. A = ложь, B = истина, C = истина; условие выполнимо
3. A = ложь, B = истина, C = ложь; условие не выполнимо
4. A = ложь, B = ложь, C = истина; условие не выполнимо

Действительно, различие между 1 и 4 тестом только в значении A, при этом результат вычисления полного условия меняется («истина» в тесте №1, «ложь» – в №4); аналогично, различие между тестами 2 и 4 только в значении B, которое изменяет результат с «истинного» в «ложное»; наконец, тесты 2 и 3 различаются только значением C, которое также изменяет результат с «истинного» в «ложное».

Таким образом, покрытие MC/DC обеспечено; отметим, что при этом критерии покрытия условий и решений также соблюдены.

Пример 2. Рассмотрим более сложный пример кода, представленный условным оператором «if ((A ∨ B) ∧ (C ∨ D))». Полная таблица истинности для условия содержит 16 строк, но для обеспечения покрытия MC/DC будет достаточно выполнения, например, только таких пяти тестов, представленных на рис.1: влияние предиката A на результат в целом проверяется тестами 1 и 4, B – соответственно 1 и 2, C – 3 и 5, D – 3 и 4.

№ теста	A	B	C	D	((A ∨ B) ∧ (C ∨ D))
1	ложь	ложь	ложь	истина	ложь
2	ложь	истина	ложь	истина	истина
3	истина	ложь	ложь	ложь	ложь
4	истина	ложь	ложь	истина	истина
5	истина	ложь	истина	ложь	истина

Рис.1. Тесты Modified condition/decision покрытия для примера 2

Необходимо отметить, что простая синтаксическая перегруппировка условий (их разбиение на несколько независимо вычисляемых с помощью временных переменных, значения которых потом используются в условии), которая не изменяет семантику программы, может понизить сложность получаемого полного MC/DC [5]. В качестве примера оценки эффективности обнаружения ошибок, можно привести

работу [6], в которой описано применение критерия MC/DC для тестирования модуля программного обеспечения спутника HETE-2.

Покрытие операторов, ветвей, MC/DC включается в сертификацию DO-178B/DO-178C безопасности оборудования авиационных систем, а также является частью требований в стандарте автомобильной безопасности ISO 26262 Road Vehicles Functional Safety.

Покрытие потока данных. Критерии покрытия потока управления (statement, branch) слишком слабы для обнаружения ошибок, а полное покрытие путей и условий может оказаться слишком трудоемким и нереализуемым на практике: фрагмент кода, имеющий *n* последовательных условий, содержит 2^n путей, а с конструкцией цикла – бесконечное множество. Покрытие потока данных служит эффективным компромиссным вариантом [3]. Рассмотрим его более подробно. Пусть G – граф потока управления тестируемой программы, G = (V, E, *s*, *f*), где V – множество вершин, E – ребер, *s* – начальная вершина, *f* – конечная. Пусть *v* – некоторая вершина потока управления из V.

Тогда пусть

defs(*v*) – множество всех переменных, которые определены в *v* (т.е. таких, которые состоят в левых частях присваиваний или каким-либо другим способом получивших новое значение);

c-use(*x*) – (префикс “c” от «computaion») – множество всех вершин, которые используют *x* для определения значения других переменных;

p-use(*x*) – (префикс “p” от «predicate») – множество всех пар вершин (v, v'), которые используют *x* в условии перехода из вершины *v* в v'.

Путь на графе G – последовательность вершин v_0, v_1, …, v_k, где для всех i, (1<i<k) ребро (v_{i-1}, v_i) принадлежит множеству E. Путь называется полным, если $v_0=s$, $v_k=f$. Пусть Π – множество путей. Будем считать, что вершина *v* включена в Π, если Π содержит путь (n_1, …, n_m), в котором $v=n_j$ для некоторого $1 \le j \le m$. Аналогично, ребро (v_1, v_2) включено в путь Π, если Π содержит путь (n_1, …, n_m), в котором $v_1=n_j$, $v_2=n_{j+1}$ для некоторого $1 \le j \le m-1$. Путь (v_1, …, v_k) включен в Π, если Π содержит путь (n_1, …, n_m), в котором $v_1=n_j$, $v_2=n_{j+1}$, …, $v_k=n_{j+k-1}$ для некоторого $1 \le j \le m-k+1$.

Назовем def-clear путем для переменной *x* такую последовательность вершин v_0, v_1, …, v_k, в которой нет присваиваний *x*, т.е. для всех i, $0 < i < k$, $x \notin$ defs(v_i).

В работе [7] Rapps и Weyuker предложили критерии для проверки всех прямых зависимостей по данным. Пусть P – множество полных путей на графе G. Тогда P удовлетворяет критерию ***All-defs***, если для каждой вершины *v* и каждой переменной $x \in$ defs(*v*), P включает соответствующий *x* def-clear путь, ведущий из *v* к, по крайней мере, одному использованию; семантически означает, что все присваивания будут как-либо задействованы. Критерий ***All-c-uses*** требует, чтобы для каждой вершины *v*, содержащей определение переменной *x*, и каждой вершины $v' \in$ c-use(*x*), достижимой из *v*, в P был покрыт def-clear путь $v..v'$; означает, что для каждого вычисления будут задействованы все определения, которые могут принимать в них участие. ***All-p-uses*** требует, чтобы для каждой вершины *v*, содержащей определение переменной *x*, и каждой пары вершин $(v', v'') \in$ p-use(*x*), в P был покрыт def-clear путь $v..v'$; означает, что все ветви будут протестированы с использованием каждого определения, составляющего их условия. ***All-c-uses/some-p-uses*** сводится к тому, что все определения задействованы и протестированы все случаи влияния на вычисления. ***All-***

p-uses/some-c-uses означает, что все определения задействованы и протестированы все случаи влияния на условия ветвей потока управления. ***All-uses*** требует, чтобы для каждого определения переменной *x* в вершине *v* в P был покрыт def-clear путь из *v* к каждой вершине из c-use(*x*) и p-use(*x*); означает, что будет протестировано каждое вычисление и условие, затронутое определением *x*. ***All-du-paths*** требует, чтобы для каждого определения переменной *x* в вершине *v*, множество тестовых путей P содержало все простые def-clear пути из *v* к каждой вершине из c-use(*x*) и p-use(*x*). ***All-paths*** требует, чтобы все пути программы были выполнены. ***All-nodes*** соответствует покрытию операторов. ***All-edges*** соответствует покрытию ветвей.

Ntafos предложил использовать стратегию требуемых k-кортежей как расширение стратегии def-use пар: в работе [8] описан критерий требуемых k-кортежей, который проверяет все цепочки k прямых зависимостей по данным, т.е. всех последовательностей из k+1 вершин, в которых каждая вершина напрямую зависима от предыдущей. Laski и Korel [9] предложили стратегию проверки зависимостей по данным в комбинациях. На рис.2 отображено взаимоотношение видов покрытий потока данных, приведенное в работе [7], расширенное критериями требуемых k-кортежей [8] и контекстного покрытия [9] (стрелками указывается включение одного (нижнего) критерия в другой (верхний); критерии на разных ветвях несравнимы). Отметим, что наиболее часто используемым критерием служит All-uses. Оценка степени покрытия затруднительна: в силу того, что не все пути выполнимы, покрытие многих def-use пар оказывается недостижимым, поэтому 100% покрытия на практике не достигается. Более того, обеспечение def-use покрытия может оказаться проблематичным для таких языковых конструкций, как массивы и указатели: нижняя аппроксимация связей может не включать некоторые достижимые def-use пары, а верхняя аппроксимация, с другой стороны, может порождать большое количество невыполнимых целей покрытия.

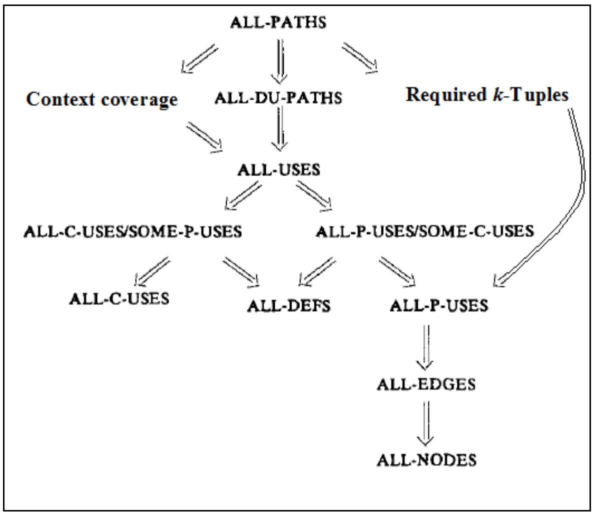


Рис.2. Иерархия типов покрытия потока данных

В работе [10] Harrold и Soffa описали анализ межпроцедурного потока данных и разработали соответствующие методы для его тестирования. Ostrand описал [11] анализ потоков данных в программах с указателями. Также в качестве примеров применения метода тестирования потока данных к Веб-приложениям можно привести работу [12], к объектно-ориентированным программам – работу [13].

Обзор алгоритмов и методов

Систематические методы генерации тестов рассматривают (необязательно конечные) абстракции пространства состояний программы, и строят тестовые сценарии для достижения искомых элементов требуемого покрытия. Такие подходы обычно реализуют некоторый вариант символьного выполнения (symbolic execution [2]) – технику анализа достижимости, которая симулирует выполнение программ над

символьными данными и вычисляет значения входов программы, удовлетворяющих условию требуемого пути.

Анализ пространства поведения систематическими методами, в отличие от анализа пространства допустимых входов, выполняемых в стохастических и комбинаторных подходах, может обнаруживать поведения, ассоциированные с очень маленьким множеством входных значений. С другой стороны, высокая вычислительная сложность такого анализа становится существенным недостатком. Производительность зависит от класса формул, которые описывают пространство состояний и производительности соответствующих солверов. Также проблемной особенностью систематических подходов является неэффективный обход пространства поведения – алгоритмы часто углубляются в перебор вариантов, не приводящий к обнаружению новых элементов искомого покрытия. В частности, большой проблемой становятся циклы, приводящие к бесконечным путям при статическом анализе потока управления. Активно разрабатываются различные эвристики для ускорения достижения покрытия, например, best-first-search [14].

Для адаптации методов к работе с исходным кодом программ, как правило, применяют методы абстракции, например, самым распространенным служит метод абстракции предикатов. В отличие от задач верификации, где строят верхнюю аппроксимацию для доказательств недостижимости, задача тестирования сводится к демонстрации достижимости, и может быть упрощена применением нижней аппроксимации, т.е. абстракция может иметь меньше поведений, чем реальная программа. Дополнительными средствами сокращения пространства поиска служат методы слоев и т.н. «воронки давления» (cone of influence reduction), идея которых заключается в элиминации тех переменных и частей программы, которые не имеют влияния на достижимость искомых элементов покрытия.

Алгоритм для автоматных моделей. Рассмотрим алгоритм [15] генерации тестовых наборов применительно для автоматных моделей с семантикой конечных транзиционных систем. На рис.3 представлен алгоритм, учитывающий частичное покрытие рассматриваемой трассой элементов, требуемых заданным критерием.

PASS:= $\emptyset$; WAIT:= $\{(s_0, A_0, C_0, \epsilon)\}$; SUITE:= $\emptyset$; COV := C_0
while WAIT$\neq \emptyset$ **do**
  select (s, A, C, ω) from WAIT; add (s, A, C, ω) to PASS
  for all $(s', A', C', \omega.t)$ **:** $(s, A, C, \omega) \overset{t}{\Rightarrow}_c (s', A', C', \omega.t)$ **do**
    if $C' \not\subseteq$ COV **then**
      add $(\omega.t, C)$ to SUITE; COV := COV $\cup\, C'$
    if $\neg\exists(s_i, A_i, C_i, \omega_i)$ **:** $(s_i, A_i, C_i, \omega_i) \in$ PASS $\cup$ WAIT $\wedge\, s_i = s' \wedge A' \subseteq A_i$ **then**
      add $(s', A', C', \omega.t)$ to WAIT
  od
od
return SUITE

Рис. 3. Алгоритм покрытия с учетом частичных элементов [15]

Алгоритм заканчивает работу, когда множество нерассмотренных состояний WAIT становится пустым. К этому моменту все достижимые состояния из начального состояния s_0 рассмотрены, множество Cov содержит все достижимые элементы покрытия, и SUITE содержит множество пар вида (w_i, C_i), где w_i – трасса, заканчивающаяся покрытием элемента C_i, и $\bigcup_i C_i$ = Cov. Алгоритм обеспечивает

полное покрытие достижимых элементов, т.к. для каждого частичного элемента a_i, расширенное состояние (s, A, C, w) такое, что $a_i \in A$ будет рассмотрено.

Использование темпоральных логик. Специфицирование целей тестирования согласно выбранному критерию возможно с помощью темпоральных логик. Такие формулы, как правило, задаются в виде «always not *p*», и называются «ловушками» (trap property). Используя различные методы проверки модели [16, 17], можно получить трассу-контрпример, нарушающую заданное свойство *p*. Например, в работе [18], Rayadurgam и Heimdahl предложили метод представления тестовых целей, обеспечивающих покрытие MC/DC, в виде формулы линейной темпоральной логики LTL, которую можно представить так:

$$\mathbf{G}(\neg(\gamma \wedge c_m \neq u_m \wedge \bigwedge_{k\neq m}(c_k = u_k))) \vee \mathbf{G}(\neg(\neg\gamma \wedge c_m = u_m \wedge \bigwedge_{k\neq m}(c_k = u_k))).$$

Здесь γ – формула полного условия, c_m – атомарный предикат, изменение значения которого должно привести к изменению выполнения полного условия γ, c_k – остальные предикаты, значения которых фиксируются, u – вектор их значений.

Для прямой зависимости по данным переменной *x*, определенной в вершине *v* и используемой в вершине v', ассоциируют LTL формулу:

$$\mathrm{ltl}(v \xrightarrow{x} v') = \mathbf{F}(v \wedge \mathbf{X}[\neg def(x)\,\mathbf{U}(v' \wedge \mathbf{F}\, f)]).$$

Таким образом, конечный путь π будет тестовой последовательностью для $v \xrightarrow{x} v'$, если, и только если, существуют такие $0 \le i < j \le k$, что $\pi(i) \models v, \pi(l) \models \neg def(x)$ для всех $i < l \le j$, $\pi(j) \models v'$ и $\pi(k) \models f$. При этом покрытие All-uses удовлетворяется множеством путей, заданных в виде $\bigcup_{v \xrightarrow{x} v'} ltl(v \xrightarrow{x} v')$.

В работе [19] зависимости по данным выражены с помощью формул WCTL (подмножества CTL, ограниченного EF, EX и EU). Например, для последовательности q=[$d_{v_1}^{x_1} u_{v_2}^{x_1} \cdot d_{v_2}^{x_2} u_{v_3}^{x_2} \cdot ... d_{v_{k-1}}^{x_{k-1}} u_{v_k}^{x_{k-1}}$], $k \ge 0$, свойства зависимости требуемых *k*-кортежей можно выразить в WCTL индуктивно:

- если *q* пустое, то *wctl*(*q*)=**EF***f*,
- если *q* в виде $[d_{v_i}^{x_i} u_{v_{i+1}}^{x_i}] \cdot q'$, то *wctl*(*q*)= $d_{v_i}^{x_i} \wedge \mathbf{EXE}[\neg def(x_i)\,\mathbf{U}(u_{v_{i+1}}^{x_i} \wedge wctl(q'))]$
- **wctl**(*q*) = **EF***wctl*(*q*)

Здесь запись d_v^x и u_v^x обозначает соответственно определение и использование переменной *x* в вершине *v*.

Динамическое символьное выполнение. Важную категорию систематического подхода составляют методы динамического символьного выполнения [20–22], известные как конкретно-символьные (concolic) и направленного случайного тестирования – гибридный подход, интегрирующий (статическое) символьное выполнение и (динамическую) информацию, полученную в процессе выполнения теста. Метод подразумевает наличие некоторого начального состояния (полученного случайным образом или заданного разработчиком), отправляясь из которого, применяет символьное выполнение, порождая новые начальные состояния для дальнейших тестов на основании условий не пройденных ветвей, встретившихся на текущем пути. В работе [23] предложен алгоритм «поиска поколений» (generational search), который существенно отличается от традиционных

стратегий поиска в глубину и в ширину: на каждой итерации рассматриваются варианты продолжения поиска вне зависимости от их длины (рис.4). Главная процедура `Search` помещает вход `inputSeed` в `workList` и выполняет тестируемую программу (строка 4). Далее обрабатывается множество `workList` (строка 5) путем выбора элемента (строка 6) и его обработки (строка 7) для генерации новых входов путем вызова функции `ExpandExecution`. Для каждого `childInputs`, тестируемая программа вызывается (строка 10), оценивается (строка 11) и добавляется в список `workList` (строка 12), который отсортирован соответственно эвристическим оценкам.

```
1  Search(inputSeed){
2    inputSeed.bound = 0;
3    workList = {inputSeed};
4    Run&Check(inputSeed);
5    while (workList not empty) {//new children
6      input = PickFirstItem(workList);
7      childInputs = ExpandExecution(input);
8      while (childInputs not empty) {
9        newInput = PickOneItem(childInputs);
10       Run&Check(newInput);
11       Score(newInput);
12       workList = workList + newInput;
13     }
14   }
15 }
```

```
1  ExpandExecution(input) {
2    childInputs = {};
3    // symbolically execute (program,input)
4    PC = ComputePathConstraint(input);
5    for (j=input.bound; j < |PC|; j++) {
6      if((PC[0..(j-1)] and not(PC[j]))
                  has a solution I){
7        newInput = input + I;
8        newInput.bound = j;
9        childInputs = childInputs + newInput;
10   }
11   return childInputs;
12 }
```

Рис.4. Алгоритм поиска поколений (generational search [23])

Главным отличием алгоритма является способ добавления потомков. Функция `ExpandExecution` символьно выполняет тестируемую программу с входом `input` и генерирует ограничение пути `PC` (строка 4). `PC` представляет собой конъюнкцию ограничений, каждая из которых соответствует условному оператору программы для текущего пути и выражена символьными переменными, представляющими значения входных параметров (см. [14, 22]). После алгоритм пытается обработать каждое ограничение пути. Это выполняется проверкой выполнимости конъюнкции части ограничений до `j`-го элемента `PC`[0..(`j`-1)] и отрицания `j`-го элемента ¬(`PC[j]`). При выполнимости, решение `I` используется для обновления предыдущего решения, таким образом, порождается условие для нового пути (строка 7). Приведенный алгоритм положен в основу ряда современных инструментальных средств, например, SAGE [24], PEX [25].

В [26] описан метод композиционного выполнения динамической генерации тестов, суть которого сводится к тестированию функций по отдельности, сохраняя результаты тестов в виде некоторого предусловия над входами и некоторого постусловия над выходами, с последующим их использованием при тестировании функций верхнего уровня. На рис.5 представлен пример, в котором при замене функции `elem_positive` ее спецификацией `(x>0 ∧ return=1) ∨ (x≤0 ∧ return=0)` количество путей при динамическом символьном выполнении сокращается с 2^N до четырех: два при обходе `elem_positive` и два при выполнении `num_positive`, при этом ограничение пути к функции `error()` будет задано формулой:

$$((m[0]>0 \wedge return_0=1) \vee (m[0]\le 0 \wedge return_0=0)) \wedge \ldots ((m[N-1]>0 \wedge return_{N-1}=1) \vee (m[N-1]\le 0 \wedge return_{N-1}=0)) \wedge (return_0+\ldots+return_{N-1}=0).$$

Аналогичная техника используется для специфицирования циклов.

`int elem_positive(int x){` `   if(x > 0) return 1;` `   return 0;` `}`	`void num_positive(int m[N]){` `   int i, res = 0;` `   for(i = 0; i < N; i++)` `      res = res + elem_positive(m[i]);` `   if(res == 0) error();` `…}`

Рис.5. Пример, демонстрирующий экспоненциальное сокращение количества путей

При выполнении программы часто необходимо обеспечить адекватную симуляцию окружающей среды. Например, в работе [27] авторы описывают использование имитационных (mock) функций для работы с базами данных. Для недетерминированных программ прибегают к тестированию «на лету», или построению тестовых сценариев в виде дерева, учитывающего возможные альтернативы поведения [17].

Конкретно-символьные генераторы используют нижнюю аппроксимацию при обработке предикатов, неразрешимых в теории используемых солверов (например, нелинейных вычислений), однако могут приводить к снижению степени покрытия.

Оптимизация тестового набора. Автоматическая генерация порождает множество тестов, которое на практике часто оказывается слишком большим, их поддержка и выполнение часто выходит за временные рамки выполнения проектов. Поиск минимального подмножества тестов, сохраняющего исходное покрытие – NP-сложная задача [28]. Из этих соображений актуальна задача эффективной *минимизации* тестового набора. Например, простая распространенная техника редукции множества тестов – элиминация префиксов, при которой проверяется, что ни один тест не является префиксом другого. Однако применение одной такой редукции недостаточно: тестовый набор все равно может содержать много избыточных тестов. В [15] применяют проверку каждого теста на предмет покрытия в нем хотя бы одного уникального по отношению к тестовому набору (без этого теста) искомого элемента критерия покрытия. Если таковых нет (т.е. все элементы в рассматриваемом тесте покрыты так или иначе в других тестах), то тест считается избыточным и удаляется из результирующего набора. В [29] представлен обзор методов минимизации тестовых наборов для регрессионного тестирования.

Для оптимизации тестовых наборов также используют такие критерии, как время выполнения, потребление ресурсов памяти, размер тестов, способность выявлять ошибки. Например, в работе [30] авторы ставят целью найти трассы, включающие максимально много элементов покрытия без сброса (reset) в исходное состояние тестируемой системы для сокращения времени выполнения тестового набора.

Проверка граничных значений позволяет повысить вероятность обнаружения ошибок [1]. В работе [31] разработан метод конкретизации символьных трасс, осуществляющий вычисление и подстановку конкретных значений, лежащих на границах допустимого диапазона.

Проблемы

Автоматически сгенерированные тесты обладают недостатками: они часто не имеют смыслового содержания; обрываются, не доводя до логического завершения (например, некоторое новое значение вычислено, но к его отображению трасса не привела, ввиду завершения по обработке несвязанной исключительной ситуации), и,

как следствие, неэффективны с точки зрения выявления ошибок (например, в [32] эксперимент с выявлением мутантов показал превосходство тестов, сгенерированных случайным образом, над кратчайшими тестами, обеспечивающими покрытие условий); последовательность событий теста обусловлена порядком перебора вариантов при обходе поведения, а не их логической взаимосвязью. В работе [33] для устранения этих недостатков использована техника пролонгации тестовых сценариев – трасса продлевается до одного из терминальных состояний модели, при этом поиск усовершенствован направляющей эвристикой, – приоритет отдается альтернативе, которая может встретить больше случаев использования (use) значений атрибутов, соответствующее присвоение (def) которых входит в обратный слайс относительно вновь покрытого элемента, что увеличивает степень причинно-следственных связей в трассе.

Автоматические тесты нуждаются в ручном сопровождении для определения условий успешного прохождения, а также для последующего обновления в связи с изменениями в коде продукта [34]. В работах [35, 36] отмечается, что плохо составленные тесты имеют негативный эффект на их сопровождение. В [37] утверждается, что пригодные к сопровождению тесты должны быть простыми, насколько это возможно, т.е. каждый тест не должен проверять слишком много функциональности, и, в то же время, избегать обфускации. Более того, накладывание тестов должно быть минимизировано, чтобы возможные изменения кода в будущем затрагивали только небольшую часть тестов. В работе [38] авторы вводят дополнительные метрики для оценки качества тестов – накладывание тестов между собой (coupling) и связность (cohesion) внутри каждого теста. Эти метрики используются в качестве весовых коэффициентов при выборе вариантов в генетическом алгоритме генерации тестов.

Главной проблемой систематических подходов остается эффект комбинаторного взрыва [3, 16, 17, 20–26]. Подходы, основанные на BDD, чувствительны к порядку атрибутов, количество вершин в худшем случае растет экспоненциально; подход Bounded Model Checking – к нетривиальным циклам, более того, в большинстве случаев на практике он не может доказать недостижимость. Рассмотрим пример программы на рис.6 (извлеченный при анализе реальных промышленных систем). Пусть требуется обеспечить покрытие операторов (отметим, что ‘`return 1`’ недостижим). Эксперименты были проведены [39] с набором различных инструментов: основанные на BDD – SMV и NuSMV; VCEGAR и BLAST реализующие метод CEGAR; основанный на SMT подход bounded model checking представлен CBMC (количество развертываний для каждого цикла было ограничено двумя итерациями); SPIN представляет explicit model checking; UPPAAL – автоматный подход; также Symbolic PathFinder (SPF) и PEX как представители symbolic execution (программа на Java для SPF была упрощена – удалены операторы ‘`goto L`’). Операционное время работы (и, соответственно,

```
int func (int p1, …, pN){
 int k1=0, …, kN=0, s=0;
 if(p1 == 1)k1 = 1;…
 if(pN == 1)kN = 1;
L:
 if(s==0  &&  k1==1){s=s+1;goto L;}
 …
 if(s==N-1 && kN==1){s=s+1;goto L;}
 if((s==N && (k1==0 ||… kN==0)) ||
    (s<N  && (k1==1 &&… kN==1))
    )return 1;
 return 0;
}
```

Рис.6. Пример программы, порождающей комбинаторный взрыв

объем требуемой памяти) всех упомянутых инструментов показали экспоненциальный рост с увеличением количества параметров, при этом ни один из них не смог обработать программу с более чем 18 параметрами.

Выводы

На пути автоматизации тестирования лежит множество технических и теоретических проблем, связанных с разрешимостью, производительностью поиска, сложными вычислениями (нелинейными, с плавающей запятой) [40] и языковыми конструкциями (например, операции со строками, указатели) [11, 41], взаимодействием тестируемых программ с окружающей средой (работа с файлами, базами данных, периферийными устройствами) [3, 27], многопоточностью, недетерминизмом и др. [17, 20].

Вопрос о способности таких тестов выявлять ошибки остается актуальным: ведь структурные критерии изначально созданы для оценки фактического покрытия кода «ручными» тестами [42]. Обычно автоматически сгенерированные по исходному коду тесты позволяют проверить только неявные требования, например, отсутствие зависаний, аварийных завершений или неожиданных исключительных ситуаций во время выполнения теста [41, 43], но не обнаружение несоответствия между реализацией системы и ее спецификациями. Для установления факта успешного прохождения теста, необходимо описать соответствующие проверки или формальную модель [44, 45] (так называемая «проблема оракула» [34]).

В [46] и [47] представлен обзор и список инструментальных средств.

Литература

1. Myers G.J. The Art Of Software Testing. N.Y. John Wiley & Sons, Inc. –2004. – 254 p.
2. King J. Symbolic execution and program testing // In Comm. of the ACM. –1976. –Vol.19. –P. 385–394.
3. Su T., Wu K., Miao W., Pu G., and oth. A Survey on Data-Flow Testing // ACM Comput. Survey. –2017. –Vol. 50(1). –35 p.
4. Ramler R., Wetzlmaier T., Klammer C. An empirical study on the application of mutation testing for a safety-critical industrial software system // Proc. of Symp. on Applied Comput. –2017. –P. 1401–1408.
5. Heimdahl M., Whalen M., Rajan A., Staats M. On MC/DC and implementation structure: An empirical study // In Proc. of Digital Avionics Systems Conf. –2008. –5 p.
6. Dupuy A., Leveson N. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software // In Proc. of Digital Aviation Systems Conf. –2000. –7 p.
7. Rapps S., Weyuker E. Data Flow Analysis Techniques for Test Data Selection // In Proc. of Int. Conf. of Software Engineering. –1982. –P. 272–277.
8. Natofos S. On Testing With Required Elements // In Proc. of COMPSAC '81, IEEE. –1981. –P.132–139.
9. Laski J., Korel B. A Data Flow Oriented Program Testing Strategy // IEEE Transactions on Software Engineering. –1983. –Vol. 9 (3). –P. 347–354.
10. Harrold M., Soffa M. Interprocedual data flow testing // In Proc. of the ACM SIGSOFT '89 Symp. on Software testing, analysis, and verification. –1989. –P. 158–167.
11. Ostrand T., Weyuker E. Data flow-based test adequacy analysis for languages with pointers // In Proc. of the Symp. on Testing, analysis, and verification. –1991. –P. 74–86.
12. Qi Y., Kung D., Wong E. An agent-based data-flow testing approach for Web applications // Information and Software Technology. –2006. –Vol. 48(12). –P. 1159–1171.
13. Denaro G., Margara A., Pezze M., Vivanti M. Dynamic data flow testing of object oriented systems // In Proc. of Int. Conf. on Software Engineering. –2015. –P. 947–958.
14. Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: a system for Automatically Generating Inputs of Death using symbolic execution // In ACM Comput. and Comm. Security. –2006. –P. 322–335.
15. Hessel A., Petterson P. A global algorithm for model-based test suite generation // Electr. Notes Theor. Comput. Sci. –2007. –Vol. 190(2). –P. 47–59.
16. Petrenko A., Silva S., Maldonado J. Model-based testing of software and systems: recent advances and challenges // Software tools for technology transfer. –2012. –Vol. 14(4). –P. 383–386.
17. Fraser G., Wotawa F., Ammann P. Testing with model checkers: a survey // Software Testing, Verification and Reliability. –2009. –Vol. 19. –P. 215–261.
18. Rayadurgam S. Heimdahl M. Coverage based test-case generation using model checkers // In Proc. of IEEE Int. Conf. on the Engineering of Computer Based Systems. –2001. –P. 83–91.
19. Hong H., Cha S., Lee I., Sokolsky O., Ural H. Data Flow Testing as Model Checking // In Proc. of Int. Conf. on Software Engineering. –2003. –P. 232–242.

20. Chen T., Zhang X. and oth. State of the art: dynamic symbolic execution for automated test generation // Future generation computer systems. –2013. –Vol. 29(7). –P.1758–1773.
21. Pasareanu C., Rungta N., Visser W. Symbolic execution with mixed concrete-symbolic solving // In Proc. of ACM Int. Sympos. of Software Testing and Analysis. –2011. –P. 35–44.
22. Godefroid O., Klarlund N., Sen K. DART: directed automated random testing // In Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation. –2005. –P. 213–223.
23. Godefroid P., Levin M., Molnar D. Automated whitebox fuzz testing // In Proc. of Network and Distributed Systems Security. –2008. –16 p.
24. Godefroid P., Levin M., Molnar D. SAGE: Whitebox Fuzzing for Security Testing //ACM Queue. –2012. –Vol. 10(1). –20 p.
25. Tillmann N., Halleux J. Pex – White Box Test Generation for .NET // In LNCS, Tests and Proofs. –2008. –Vol. 4966. –P. 134–153.
26. Christakis M., Godefroid P. IC-Cut: A Compositional Search Strategy for Dynamic Test Generation // In Proc. of Int. Symp. SPIN 2015 on Model Checking Software. –2015. –Vol. 9232. –P. 300–318.
27. Teneja K., Zhang Y., Xie T. MODA: automated test generation for database applications via mock objects // In Proc. of Int. Conf. on Automated Software Engineering. –2010. –P. 289–292.
28. Karp R. Reducibility among combinatorial problems // Complexity of computer computations. –1972. –P. 85–103.
29. Yoo S., Harman M. Regression testing minimization, selection and prioritization: a survey // Software Testing, Verification & Reliability. –2012. –Vol. 22(2). –P. 67–120.
30. Schrammel P., Melham T., Kroening D. Generating test case chains for reactive systems // Int. J. Softw. Tools Technology Transfer. –2016. –Vol. 16(3). –P. 319–334.
31. Kolchin A., Letichevsky A., Peschanenko V., Drobintsev P., Kotlyarov V. An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. –2013. –Vol. 47(7). –P. 433–442.
32. Heimdahl M., Devaraj G., Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? // In IEEE Computer society, HASE. –2004. –P. 178–186.
33. Колчин А.В., Потиенко С.В. Метод генерации тестовых данных по исходному коду Java программ // Искусственный интеллект. –2016. –№3. –С. 50–58.
34. Barr E., Harman M., McMinn P., Shahbaz M., Yoo S. The oracle problem in software testing: a survey // IEEE Transactions on Software Engineering. –2015. –Vol. 41. –P. 507–525.
35. Athanasiou D., Nugroho A., Visser J. and Zaidman A. Test code quality and its relation to issue handling performance // IEEE Transactions on Software Engineering. –2014. –Vol. 40(11). –P. 1100–1125.
36. Rojas J., Fraser G., Arcuri A. Automated unit test generation during softweare development: a controlled experiment and thing-aloud observations. // In proc. Of the Int. Sympos. on Software Testing and Analysis. –2015. –P. 338–349.
37. Meszaros G. XUnit Test Patterns: refactoring test code. Addison-Wesley Professional. –2007. –944 p.
38. Palomba F., Panichella A., Zaidman A., Oliveto R., Lucia A. Automatic Test Case Generation: What if Test Code Quality Matters? // In Proc. of Int. Symp. on Softw. Testing and Analysis. –2016. –P. 130–141.
39. Kolchin A. About efficiency problems of reachability proving using model checking approach // In Proc. of Int. Conf. on Computational Intelligence (results, problems and perspectives). –2017. –P. 95–96.
40. Braione P., Denaro G., Mattavelli A., Vivanti M., Muhammad A. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component // Software Quality Journal. –2014. –P. 311–333.
41. Cseppento L., Micskei Z. Evaluating symbolic execution-based test tools // In IEEE Int. Conf. on Software Testing, Verification and Validation. –2015. –P. 1–10.
42. Devaraj G., Heimdahl M., Liang D. Coverage-directed test generation with model checkers: challenges and opportunities // In Proc. of Int. Conf. COMPSAC'2005. –2005. –Vol. 1. –P. 455–462.
43. Fraser G. Arcuri A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite // Emperical software engineering. –2015. –Vol. 20(3). –P. 611–639.
44. Letichevsky A., Letychevskyi O., Peschanenko V. Symbolic Modelling in White-Box Model-Based Testing // In Proc. of Int. Conf. on Artificial Intelligence, Modelling and Simulation. –2015. –P. 237–240.
45. Letychevskyi O. Paradigms of Model-Based and Symbolic Testing of Software Systems // Cybernetics and Systems Analysis. –2015. –V. 51. –N 5. –P. 692–703.
46. Galler S., Aichernig B. Survey on test data generation tools // Int. Journal on Software Tools for Technology Transfer. –2014. –Vol. 16(6). –P. 727–751.
47. [Электронный ресурс]. –Режим доступа: http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html

Literatura

1. Myers G.J. The Art Of Software Testing. N.Y. John Wiley & Sons, Inc. –2004. – 254 p.
2. King J. Symbolic execution and program testing // In Comm. of the ACM. –1976. –Vol.19. –P. 385–394.
3. Su T., Wu K., Miao W., Pu G., and oth. A Survey on Data-Flow Testing // ACM Comput. Survey. –2017. –Vol. 50(1). –35 p.

4. Ramler R., Wetzlmaier T., Klammer C. An empirical study on the application of mutation testing for a safety-critical industrial software system // Proc. of Symp. on Applied Comput. –2017. – P. 1401–1408.
5. Heimdahl M., Whalen M., Rajan A., Staats M. On MC/DC and implementation structure: An empirical study // In Proc. of Digital Avionics Systems Conf. –2008. –5 p.
6. Dupuy A., Leveson N. An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software // In Proc. of Digital Aviation Systems Conf. –2000. –7 p.
7. Rapps S., Weyuker E. Data Flow Analysis Techniques for Test Data Selection // In Proc. of Int. Conf. of Software Engineering. –1982. –P. 272–277.
8. Natofos S. On Testing With Required Elements // In Proc. of COMPSAC '81, IEEE. –1981. –P.132–139.
9. Laski J., Korel B. A Data Flow Oriented Program Testing Strategy // IEEE Transactions on Software Engineering. –1983. –Vol. 9 (3). –P. 347–354.
10. Harrold M., Soffa M. Interprocedual data flow testing // In Proc. of the ACM SIGSOFT '89 Symp. on Software testing, analysis, and verification. –1989. –P. 158–167.
11. Ostrand T., Weyuker E. Data flow-based test adequacy analysis for languages with pointers // In Proc. of the Symp. on Testing, analysis, and verification. –1991. –P. 74–86.
12. Qi Y., Kung D., Wong E. An agent-based data-flow testing approach for Web applications // Information and Software Technology. –2006. –Vol. 48(12). –P. 1159–1171.
13. Denaro G., Margara A., Pezze M., Vivanti M. Dynamic data flow testing of object oriented systems // In Proc. of Int. Conf. on Software Engineering. –2015. –P. 947–958.
14. Cadar C., Ganesh V., Pawlowski P., Dill D., Engler D. EXE: a system for Automatically Generating Inputs of Death using symbolic execution // In ACM Comput. and Comm. Security. –2006. –P. 322–335.
15. Hessel A., Petterson P. A global algorithm for model-based test suite generation // Electr. Notes Theor. Comput. Sci. –2007. –Vol. 190(2). –P. 47–59.
16. Petrenko A., Silva S., Maldonado J. Model-based testing of software and systems: recent advances and challenges // Software tools for technology transfer. –2012. –Vol. 14(4). –P. 383–386.
17. Fraser G., Wotawa F., Ammann P. Testing with model checkers: a survey // Software Testing, Verification and Reliability. –2009. –Vol. 19. –P. 215–261.
18. Rayadurgam S., Heimdahl M. Coverage based test-case generation using model checkers // In Proc. of IEEE Int. Conf. on the Engineering of Computer Based Systems. –2001. –P. 83–91.
19. Hong H., Cha S., Lee I., Sokolsky O., Ural H. Data Flow Testing as Model Checking // In Proc. of Int. Conf. on Software Engineering. –2003. –P. 232–242.
20. Chen T., Zhang X. and oth. State of the art: dynamic symbolic execution for automated test generation // Future generation computer systems. –2013. –Vol. 29(7). –P. 1758–1773.
21. Pasareanu C., Rungta N., Visser W. Symbolic execution with mixed concrete-symbolic solving // In Proc. of ACM Int. Sympos. of Software Testing and Analysis. –2011. –P. 35–44.
22. Godefroid O., Klarlund N., Sen K. DART: directed automated random testing // In Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation. –2005. –P. 213–223.
23. Godefroid P., Levin M., Molnar D. Automated whitebox fuzz testing // In Proc. of Network and Distributed Systems Security. –2008. –16 p.
24. Godefroid P., Levin M., Molnar D. SAGE: Whitebox Fuzzing for Security Testing //ACM Queue. –2012. –Vol. 10(1). –20 p.
25. Tillmann N., Halleux J. Pex – White Box Test Generation for .NET // In LNCS, Tests and Proofs. –2008. –Vol. 4966. –P. 134–153.
26. Christakis M., Godefroid P. IC-Cut: A Compositional Search Strategy for Dynamic Test Generation // In Proc. of Int. Symp. SPIN 2015 on Model Checking Software. –2015. –Vol. 9232. –P. 300–318.
27. Teneja K., Zhang Y., Xie T. MODA: automated test generation for database applications via mock objects // In Proc. of Int. Conf. on Automated Software Engineering. –2010. –P. 289–292.
28. Karp R. Reducibility among combinatorial problems // Complexity of computer computations. –1972. –P. 85–103.
29. Yoo S., Harman M. Regression testing minimization, selection and prioritization: a survey // Software Testing, Verification & Reliability. –2012. –Vol. 22(2). –P. 67–120.
30. Schrammel P., Melham T., Kroening D. Generating test case chains for reactive systems // Int. J. Softw. Tools Technology Transfer. –2016. –Vol. 16(3). –P. 319–334.
31. Kolchin A., Letichevsky A., Peschanenko V., Drobintsev P., Kotlyarov V. An approach to creating concretized test scenarios within test automation technology for industrial software projects // Automatic Control and Computer Sciences. –2013. –Vol. 47(7). –P. 433–442.
32. Heimdahl M., Devaraj G., Weber R. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? // In IEEE Computer society, HASE. –2004. –P. 178–186.
33. Kolchin A.V., Potienko S.V. Metod generatsii testovyih dannyih po ishodnomu kodu Java programm // Iskusstvennyiy intellekt. –2016. –#3. –S. 50–58.
34. Barr E., Harman M., McMinn P., Shahbaz M., Yoo S. The oracle problem in software testing: a survey // IEEE Transactions on Software Engineering. –2015. –Vol. 41. –P. 507–525.

35. Athanasiou D., Nugroho A., Visser J., Zaidman A. Test code quality and its relation to issue handling performance // IEEE Transactions on Software Engineering. –2014. –Vol. 40(11). –P. 1100–1125.
36. Rojas J., Fraser G, Arcuri A. Automated unit test generation during softweare development: a controlled experiment and thing-aloud observations. // In proc. Of the Int. Sympos. on Software Testing and Analysis. –2015. –P. 338–349.
37. Meszaros G. XUnit Test Patterns: refactoring test code. Addison-Wesley Professional. –2007. –944 p.
38. Palomba F., Panichella A., Zaidman A., Oliveto R., Lucia A. Automatic Test Case Generation: What if Test Code Quality Matters? // In Proc. of Int. Symp. on Softw. Testing and Analysis. –2016. –P. 130–141.
39. Kolchin A. About efficiency problems of reachability proving using model checking approach // In Proc. of Int. Conf. on Computational Intelligence (results, problems and perspectives). –2017. –P. 95–96.
40. Braione P., Denaro G., Mattavelli A., Vivanti M., Muhammad A. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component // Software Quality Journal. –2014. –P. 311–333.
41. Cseppento L., Micskei Z. Evaluating symbolic execution-based test tools // In IEEE Int. Conf. on Software Testing, Verification and Validation. –2015. –P. 1–10.
42. Devaraj G., Heimdahl M., Liang D. Coverage-directed test generation with model checkers: challenges and opportunities // In Proc. of Int. Conf. COMPSAC'2005. –2005. –Vol. 1. –P. 455–462.
43. Fraser G., Arcuri A. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with Evosuite // Emperical software engineering. –2015. –Vol. 20(3). –P. 611–639.
44. Letichevsky A., Letychevskyi O., Peschanenko V. Symbolic Modelling in White-Box Model-Based Testing // In Proc. of Int. Conf. on Artificial Intelligence, Modelling and Simulation. –2015. –P. 237–240.
45. Letychevskyi O. Paradigms of Model-Based and Symbolic Testing of Software Systems // Cybernetics and Systems Analysis. –2015. –V. 51. –N 5. –P. 692–703.
46. Galler S., Aichernig B. Survey on test data generation tools // Int. Journal on Software Tools for Technology Transfer. –2014. –Vol. 16(6). –P. 727–751.
47. [Elektronnyiy resurs]. –Rezhim dostupa:: http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html

RESUME

V. Volkov, A. Kolchin, A. Letychevskyy, S. Potiyenko

A survey of systematic methods for code-based test data generation

The paper presents a survey of prevailing code coverage types, including modified condition/decision and dataflow coverage, and modern automatic test data generation methods based on source code of imperative programs.

A number of small examples are given to illustrate addressed coverage types; dataflow coverage description is based on Rapps and Weyuker's definitions of def-use pairs for direct data dependencies checking, and enhancements provided by Ntafos (k-tuples), Laski and Korel (combination strategy).

Different ways of testing goals specification in terms of the automaton model and temporal logics are described. Special emphasis is made for systematic methods, which are based on the concolic (a portmanteau of concrete and symbolic) and symbolic execution techniques – constraint solving, symbolic model checking, and dynamic symbolic execution. Algorithms of generational search and global coverage for test suite generation with partial coverage items are described. Various practices of test set optimization with respect to size, execution time, quality and ability to cover boundary values are surveyed.

An example of a small program which forces to exponential growth of time and memory while statement coverage achieving is described. The example is tested on several different up-to-date instrumental tools: SMV, NuSMV, VCEGAR, BLAST, CBMC, SPIN, UPPAAL, Symbolic PathFinder and PEX.

The survey discusses actual problems of the automatic test generation approach, such as combinatorial explosion of the number of possible program paths, minimization of the resulting test cases set, its maintainability, an estimation of its quality, coupling and cohesion, ability to errors revealing and common types of automatically detected defects.

Надійшла до редакції 23.10.2017

